210CT Coursework

GITHUB LINK - <https://github.com/nanrak/210CT>

Task 1 - Build a Binary Search Tree (BST) to hold English language words in its nodes. Use a paragraph (1 paragraph would suffice) of any text in order to extract words and to determine their frequencies. Input: You should read the paragraph from a suitable file format, such as .txt. The following tree operations should be implemented: a) printing the tree in pre-order. b) Finding a word. Regardless whether found or not found, your program should output the path traversed in determining the answer, followed by yes if found or no, respectively. Note: In order to deal with identical words, add another attribute to the Binary Search Tree node that would keep track of the frequency of that word.

class Node:

def \_\_init\_\_(self,word):

self.word = word

self.freq = 1

self.left = None

self.right = None

def \_\_str\_\_(self):

return self.word + ' ' + str(self.freq)

class searchtree:

def \_\_init\_\_(self):

self.root = None

def create(self,val):

if self.root == None:

self.root = Node(val)

else:

current = self.root

while 1:

if val < current.word:

if current.left:

current = current.left

else:

current.left = Node(val)

break;

elif val > current.word:

if current.right:

current = current.right

else:

current.right = Node(val)

break;

else:

current.freq += 1

break

def preorder(self,node):

if node is not None:

print (node)

self.preorder(node.left)

self.preorder(node.right)

def find(self,search) :

if self.root == None:

print 'Tree is empty'

else:

current = self.root

while 1:

print current

if search == current.word:

print 'Yes'

break;

elif search < current.word:

if current.left:

current = current.left

else:

print 'No'

break;

elif search > current.word:

if current.right:

current = current.right

else:

print 'No'

break;

##################################################################################

tree = searchtree()

file = open('paragraph.txt', 'r')

for word in file.read().split():

tree.create(word)

file.close()

print 'Preorder Traversal\n=================='

tree.preorder(tree.root)

print '\n'

print 'Search for word \'world\'\n======================='

tree.find('world')

print '\n'

print 'Search for word \'over\'\n======================'

tree.find('over')

Task 2 - Implement in the language of your choice a function that deletes a node in a Binary Search Tree. Integrate this function in the code provided to you on Moodle. The deletion operation should be performed based on the key (value) of the node. It is up to you how you print the binary search tree - you can just use a traversal method such as in-order or you can create a fancier, more visual representation.

class BinTreeNode(object):

def \_\_init\_\_(self, value):

self.value=value

self.left=None

self.right=None

def tree\_insert(tree, item):

if tree==None:

tree=BinTreeNode(item)

else:

if(item < tree.value):

if(tree.left==None):

tree.left=BinTreeNode(item)

else:

tree\_insert(tree.left,item)

else:

if(tree.right==None):

tree.right=BinTreeNode(item)

else:

tree\_insert(tree.right,item)

return tree

def tree\_delete(tree, item):

if item < tree.value:

tree.left = tree\_delete(tree.left, item)

elif(item > tree.value):

tree.right = tree\_delete(tree.right, item)

else:

if tree.left is None :

temp = tree.right

tree = None

return temp

elif tree.right is None :

temp = tree.left

tree = None

return temp

temp = tree.right

while(temp.left is not None):

temp = temp.left

tree.value = temp.value

tree.right = tree\_delete(tree.right , temp.value)

return tree

def postorder(tree):

if(tree.left!=None):

postorder(tree.left)

if(tree.right!=None):

postorder(tree.right)

print (tree.value)

def in\_order(tree):

if(tree.left!=None):

in\_order(tree.left)

print (tree.value)

if(tree.right!=None):

in\_order(tree.right)

##################################################################################

if \_\_name\_\_ == '\_\_main\_\_':

t = tree\_insert(None,6)

tree\_insert(t,10)

tree\_insert(t,5)

tree\_insert(t,2)

tree\_insert(t,3)

tree\_insert(t,4)

tree\_insert(t,11)

print 'Inorder Traversal\n================='

in\_order(t)

t = tree\_delete(t,3)

t = tree\_delete(t,6)

print '\nAfter deleting 3 and 6\n======================'

in\_order(t)

Task 3 - Implement an unweighted and undirected graph data structure in the programming language of your choice, where the nodes consist of positive integers. You can either use an adjacency matrix or an adjacency list approach. The program should have functions for the following: a. adding a node to the graph. b. Adding an edge to the graph. c. Printing the graph. Note: You must use Object Oriented Programming for this task.

class Node:

def \_\_init\_\_(self, name):

self.name = name

self.neighbors = []

def add\_neighbor(self, neighbor):

if neighbor.name not in self.neighbors:

self.neighbors.append(neighbor.name)

neighbor.neighbors.append(self.name)

self.neighbors = sorted(self.neighbors)

neighbor.neighbors = sorted(neighbor.neighbors)

def \_\_repr\_\_(self):

return str(self.neighbors)

class Graph:

def \_\_init\_\_(self):

self.nodes = {}

def add\_node(self, node):

self.nodes[node.name] = node.neighbors

def add\_edge(self, node\_from, node\_to):

node\_from.add\_neighbor(node\_to)

self.nodes[node\_from.name] = node\_from.neighbors

self.nodes[node\_to.name] = node\_to.neighbors

def adjacency\_list(self):

if len(self.nodes) >= 1:

return [str(key) + ':' + str(self.nodes[key]) for key in self.nodes.keys()]

else:

return dict()

def \_\_str\_\_(self):

return str(self.adjacency\_list())

##################################################################################

a = Node('A')

b = Node('B')

c = Node('C')

d = Node('D')

e = Node('E')

g = Graph()

g.add\_node(a)

g.add\_node(b)

g.add\_node(c)

g.add\_node(d)

g.add\_node(e)

g.add\_edge(a,b)

g.add\_edge(a,c)

g.add\_edge(b,c)

g.add\_edge(b,d)

g.add\_edge(c,e)

g.add\_edge(d,b)

g.add\_edge(e,d)

print g

Task 4/ 5 – Using the graph structure implemented last week, implement a function isPath(v, w), where v and w are vertices in the graph, to check if there is a path between the two nodes. The path found will be printed to a text file as a sequence of integer numbers (the node values).

Using the same graph structure, implement a function isConnected(G) to check whether or not the graph is connected. The output should be simply 'Yes' or 'No'.

Implement the BFS and DFS traversals for the above graph structure. Save the nodes traversed for each of the two traversals in sequence to a text file.

class Graph:

def \_\_init\_\_(self):

self.nodes = {}

def add\_node(self, node):

self.nodes[node.name] = node.neighbors

def add\_edge(self, node\_from, node\_to):

node\_from.add\_neighbor(node\_to)

self.nodes[node\_from.name] = node\_from.neighbors

self.nodes[node\_to.name] = node\_to.neighbors

def isPath(self, v, w, path=[]):

path = path + [v]

if v == w:

return path

if v not in self.nodes:

return None

for node in self.nodes[v]:

if node not in path:

newpath = self.isPath(node, w, path)

if newpath:

return newpath

return None

def adjacency\_list(self):

if len(self.nodes) >= 1:

return [str(key) + ':' + str(self.nodes[key]) for key in self.nodes.keys()]

else:

return dict()

def \_\_str\_\_(self):

return str(self.adjacency\_list())

def isConnected(g):

is\_path = True

nodes = g.nodes.keys()

n = len(nodes)

for i in range(0, n):

for j in range(i+1, n):

path = g.isPath(nodes[i], nodes[j])

if path == None:

is\_path = False;

break;

if not is\_path:

break;

print 'Yes' if is\_path else 'No'

def bfs(g):

path=[]

start = g.nodes.keys()[0]

q = [start]

while q:

v = q.pop(0)

if not v in path:

path = path + [v]

q = q + g.nodes[v]

return path

def dfs(g):

path=[]

start = g.nodes.keys()[0]

q = [start]

while q:

v = q.pop(0)

if v not in path:

path = path + [v]

q = g.nodes[v] + q

return path

##################################################################################

a = Node(1)

b = Node(2)

c = Node(3)

d = Node(4)

e = Node(5)

f = Node(6)

g = Graph()

g.add\_node(f)

g.add\_node(b)

g.add\_node(a)

g.add\_node(c)

g.add\_node(e)

g.add\_node(d)

g.add\_edge(a,b)

g.add\_edge(a,c)

g.add\_edge(b,d)

g.add\_edge(c,e)

g.add\_edge(c,f)

print g

path = g.isPath(a.name,d.name)

if path != None:

file = open(str(a.name) + '\_' + str(d.name) + '\_path.txt', 'w')

file.write(str(path))

file.close()

isConnected(g)

x = Node(7)

g.add\_node(x)

print g

isConnected(g)

path = bfs(g)

print path

file = open('bfs.txt', 'w')

file.write(str(path))

file.close()

path = dfs(g)

print path

file = open('dfs.txt', 'w')

file.write(str(path))

file.close()

Task 6 - Adapt your graph structure from the previous week to support weights for the edges and implement Dijkstra’s algorithm. The output of the program should be the total cost and the actual path found.

import sys

class Node:

def \_\_init\_\_(self, label):

self.label = label

class Edge:

def \_\_init\_\_(self, to\_node, weight):

self.to\_node = to\_node

self.weight = weight

class Graph:

def \_\_init\_\_(self):

self.nodes = set()

self.edges = dict()

def add\_node(self, node):

self.nodes.add(node)

def add\_edge(self, from\_node, to\_node, weight):

edge = Edge(to\_node, weight)

if from\_node.label in self.edges:

from\_node\_edges = self.edges[from\_node.label]

else:

self.edges[from\_node.label] = dict()

from\_node\_edges = self.edges[from\_node.label]

from\_node\_edges[to\_node.label] = edge

def min\_cost(q, cost):

min\_node = None

for node in q:

if min\_node == None:

min\_node = node

elif cost[node] < cost[min\_node]:

min\_node = node

return min\_node

def dijkstra(graph, start):

q = set()

cost = {}

prev = {}

for v in graph.nodes:

cost[v] = sys.maxint

prev[v] = sys.maxint

q.add(v)

cost[start] = 0

while q:

v = min\_cost(q, cost)

q.remove(v)

if v.label in graph.edges:

for \_, w in graph.edges[v.label].items():

temp = cost[v] + w.weight

if temp < cost[w.to\_node]:

cost[w.to\_node] = temp

prev[w.to\_node] = v

return cost, prev

def path(prev, from\_node):

previous\_node = prev[from\_node]

route = [str(from\_node.label)]

while previous\_node != sys.maxint:

route.append(str(previous\_node.label))

temp = previous\_node

previous\_node = prev[temp]

route.reverse()

return route

##################################################################################

graph = Graph()

a = Node(1)

graph.add\_node(a)

b = Node(2)

graph.add\_node(b)

c = Node(3)

graph.add\_node(c)

d = Node(4)

graph.add\_node(d)

e = Node(5)

graph.add\_node(e)

f = Node(6)

graph.add\_node(f)

g = Node(7)

graph.add\_node(g)

graph.add\_edge(a, b, 4)

graph.add\_edge(a, c, 3)

graph.add\_edge(a, e, 7)

graph.add\_edge(b, c, 6)

graph.add\_edge(b, d, 5)

graph.add\_edge(c, d, 11)

graph.add\_edge(c, e, 8)

graph.add\_edge(d, e, 2)

graph.add\_edge(d, f, 2)

graph.add\_edge(d, g, 10)

graph.add\_edge(e, f, 5)

graph.add\_edge(f, g, 3)

cost, prev = dijkstra(graph, a)

print 'path : ' + str(path(prev, g))

print 'cost : ' + str(cost[g])